How do I get the effect of C#'s async void in a C++ coroutine? Part 3: Simplifying the boilerplate

devblogs.microsoft.com/oldnewthing/20190118-00

January 18, 2019

Raymond Chen

Last time, we figured out how to use a coroutine in a place where the caller expects a function returning `void`. It required some wrapping, and our research led to this pattern:

```
void MyClass::MyEventHandler(int a, int b)
{
  [](auto lambda1)
   -> Concurrency::task<void>
  {
   co_await lambda1();
  }([=, lifetime = std::shared_from_this(this)]()
   -> Concurrency::task<void>
  {
   // actual work goes here
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  });
}
```

You might think "Maybe I can macro-ize this thing so I don't have to repeat the boilerplate all the time."

```
#define INVOKE_ASYNC_LAMBDA(lambda) \
  [](auto lambda1) \
   -> Concurrency::task<void> \
  { \
   co_await lambda1(); \
  }(lambda)

void MyClass::MyEventHandler(int a, int b)
{
  INVOKE_ASYNC_LAMBDA(
   [=, lifetime = std::shared_from_this(this)]()
   -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  });
}
```

But then you realize that you've gone too far, because you've created a macro that requires people to pass a lambda as a macro parameter, and that road leads to sadness.

So you might wrack your brains for a while to see if there's a way to get the boilerplate code generated without requiring the lambda as a macro parameter. Maybe something like this:

```
#define INVOKE_ASYNC_LAMBDA \
  [](auto lambda1) \
   -> Concurrency::task<void> \
  { \
   co_await lambda1(); \
  }
```

Since all we do with the lambda is spit it back out, including parentheses, and the regurgitation is as the very last tokens of the macro expansion we can cheat and avoid capturing the parameter at all. The macro spits out the boilerplate, and then what looks like the argument to the macro is actually just text that comes after the macro, and a parenthesized lambda happens to be exactly what we want to come next, so *jackpot*.

But then you remember the C++ Core Guidelines which says,

> Scream when you see a macro that isn't just used for source control (e.g., `#ifdef`)

Is there a way to do this that avoids macros entirely?

Indeed there is, but you have to back up a step. The step prior to our "final" version went like this:

```
void MyClass::MyEventHandler(int a, int b)
{
  auto lambda2 = [](auto lambda1)
   -> Concurrency::task<void>
  {
   co_await lambda1();
  };

  lambda2([=, lifetime = std::shared_from_this(this)]()
   -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  });
}
```

The captureless lambda can be factored out into a templated free function.

```
template<typename TLambda>
Concurrency::task<void>
invoke_async_lambda(TLambda lambda)
{
  co_await lambda();
}

void MyClass::MyEventHandler(int a, int b)
{
  invoke_async_lambda(
  [=, lifetime = std::shared_from_this(this)]()
   -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  });
}
```

And then we can generalize the function further by having it return the same type of task that the lambda does.[1]

```
template<typename TLambda>
auto invoke_async_lambda(TLambda lambda)
 -> decltype(lambda())
{
  co_return co_await lambda();
}
```

Now you can use it for async lambdas that return any kind of awaitable object, like a `Concurrency::task<int>`, or a `winrt::Windows::Foundation::IAsyncAction`, or a `std::experimental::future<std::string>`. And since it returns the

resulting coroutine, you can continue operating with it.

```
Concurrency::task<void>
DoThreeThingsAsync()
{
 std::array<Concurrency::task<void>, 3> tasks =
 {
  invoke_async_lambda([=] -> Concurrency::task<void>
  {
   ... first task ...
  }),
  invoke_async_lambda([=] -> Concurrency::task<void>
  {
   ... second task ...
  }),
  invoke_async_lambda([=] -> Concurrency::task<void>
  {
   ... third task ...
  })
 };

 return Concurrency::when_all(begin(tasks), end(tasks));
}
```

[1] The result is one of those cryptic functions that doesn't seem to do anything, but in fact does quite a bit, but in a very subtle way. The C++ standard library has a lot of functions like that, such as `std::move`, `std::forward`, and `std::launder`.

Raymond Chen

Follow